DSA Homework 4: Recursive Algorithms

David Tarazi

February 21, 2020

1 Eating Sequentially

Although I like the thought, I don't think this will be an effective strategy for students to eat. Nevertheless, there is a recursive way to find the maximum happiness based on food liking in sequential order. Given a list of happiness values in order from how a student may select the items, you have a function, *getMax(list, max, bestList)*, that will take in a list, the current max happiness, and the current best list. The function loops through the list to calculate the sum after each number and will check if it is greater than the current max happiness, it will save the best list and its max sum. After looping through all the possible combinations containing the full list, you run recursively run the same function with the list while chopping off the last value of the list with each iteration until you reach the base case of a list with length = 1. You have then run through all the possible combinations of sequential sublists within the larger list which will allow you to find the correct max value and interval.

To prove correctness, every time you run the function, you check each combination of sequential values ending with the last value of the list. As you shrink the length by chopping off the last food item until there are no more food items left, you are finding what the happiness would be if you start at every food item until the last one as it shrinks. As a result, you have found the sum of happiness for each and every combination of food options giving you the maximum happiness. Pseudocode shown below.

```
maxSum = 0
bestList = []
getMax(list, maxSum, bestList):
    if len = 1: return bestList
    sum = 0
    for i in range(len(list)):
        sum += list[-i]
        if sum > maxSum:
            maxSum = sum
            bestList = list[-i:end]
    return getMax(list[0:end-1], maxSum, bestList)
```

The runtime of this implementation will be O(n^2) since the function will have to run n times as it shrinks by 1 until it reaches a length of 1 starting from n and every function call loops through each element of the list so you essentially have a nested O(n) runtime in the O(n) function resulting in O(n^2). To maybe speed it up, you could break the loop if the last element is negative to skip it in the same way that you would skip food you dislike without considering it.

2 Acedemitis!

There are a couple of things that we need to keep track of as shown below:

- Which students have been in class with a potentially infected student (list)

- Which students are already in the ”could have the disease” pool (flag)

By tracking these values, you could use a queue to store all the ”potentially infected” students by looping through the list of patient 0’s classmates and for each classmate, we loop down their branch of classmates and their classmate’s classmates and so on. For every person, we check if they have already been placed in the queue and if not, we add them to the queue of ”potentially infected” students and flag them as an ”already infected” student. We break out of a single branch once there is no one who isn’t already in the ”potentially infected” queue and then go on to the next classmate of patient 0. By iterating through all of the classmates and their entire branch while checking if they have been infected already. As a result, we are iterating through every person’s classmates from beginning to end and flagging them as we iterate through all possible options. Although runtime may be long, there isn’t a single classmate that isn’t checked and therefore the queue will hold all the possibly infected students.

3 Implementation

See dtarazi.py